Bank Customer Churn Analysis And Prediction – Project Exploration & Code Breakdown

1. Source Code

This project uses the Kaggle notebook below as a reference to understand the churn analysis steps and model building process: https://www.kaggle.com/code/aliaagamal/bank-customer-churn-analysis-and-prediction#4.-Model-Implementation

2. Project & Data Explanation

a. Objective

- This project aims to predict whether a bank customer will leave the bank (churn) or stay, based on various customer attributes.
- Customer churn is a critical issue for banks, as retaining existing customers is significantly more cost-effective than acquiring new ones.
- Understanding the factors that influence churn allows banks to design targeted loyalty programs and more effective retention strategies.
- The goal of this project is to analyze these factors and build a machine learning model capable of identifying customers who are at risk, enabling better decision-making for customer retention and intervention planning.

b. Dataset Description

The dataset contains 18 columns, each representing a feature of a bank customer. These features include demographic details, financial information, customer engagement, and feedback metrics. Below is a detailed description of the columns, grouped by their relevance for better understanding:

1. Identifiers

Column Name	Description
RowNumber	A sequential number for each record.
CustomerId	A unique identifier for each customer.
Surname	The customer's last name.

2. Demographic Information

Column Name	Description
Geography	The country or region of the customer.
Gender	Customer gender (Male/Female).
Age	Customer age. Older customers are generally more loyal.

3. Financial Information

Column Name	Description
CreditScore	Customer credit score (300–850). Higher score = lower churn risk.
Balance	Account balance. Higher balances often correlate with lower churn.
EstimatedSalary	Estimated customer salary.

4. Customer Engagement

Column Name	Description
Tenure	Years the customer has stayed with the bank.
NumOfProducts	Number of bank products used.
HasCrCard	Whether the customer has a credit card.
IsActiveMember	Whether the customer is an active member.
Card Type	Type of credit card used.
Points Earned	Points collected from card usage.

5. Customer Feedback

Column Name	Description
Complain	Whether the customer has filed a complaint.
Satisfaction Score	Score given for complaint resolution.

6. Target Variable

Column Name	Description
Exited	Whether the customer left the bank (0 = Stayed, 1 = Left).

c. Business problem

- **Primary Goal:** To mitigate customer attrition (churn) which directly impacts the bank's revenue.
- **High Acquisition Cost:** Acquiring new customers is significantly more expensive (5 to 25 times) than retaining existing ones.
- **Financial Impact:** Churn leads to substantial revenue loss and a decrease in Customer Lifetime Value (CLV).
- **Strategic Need:** The bank requires a shift from a reactive approach to a predictive/proactive strategy to identify high-risk customers.
- **Project Solution:** To build an accurate predictive model to determine which customers are likely to churn, allowing retention campaigns to be deployed in a targeted and efficient manner.

3. Project Workflow

a. Step-by-step:

1. **Collect & load dataset:** To load data in local CSV file, make sure The CSV file(Customer-Churn-Records.csv) placed in the same directory as your Python script or Jupyter Notebook.
2. **Data Exploration:** Taking an initial look at the dataset's random sample data, basic structure, size, data types, and checking for missing or inconsistent and duplicate values.
3. **Exploratory Data Analysis (EDA):** Deeper investigation using statistics and visualizations (charts/graphs) to understand patterns, relationships between features, and the distribution of the target variable (e.g., Churn).
4. **Data Preprocessing and Cleaning:** Transforming the data to make it suitable for the model. This includes handling outliers and encoding categorical variables (like turning 'Male'/'Female' into numbers).
5. **Train-test split:** Dividing the clean data into three unique sets: Training (to teach the model), Validation (to tune model hyperparameters and compare different models), and Testing (for a final, unbiased performance check).
6. **Model Implementation:** Selecting and building an appropriate ML algorithm (like Random Forest), training it with the Training data, and refining its performance using the Validation data.
7. **Model saving (pkl):** Storing the final, trained, and optimal model object into a file (usually the .pkl format) so it can be used anytime without needing to be retrained.

8. **Load model + generate predictions:** Loading the saved model file from storage and using it to make real-world forecasts or predictions on new, current customer data.

b. Environment / libraries used

- **Data Manipulation**
  - *pandas:* Main tool for working with table data (DataFrames); loading and manipulating data efficiently.
    Parameters (Example: pd.read_csv()):
    a. *filepath_or_buffer* (where the data is),
    b. *sep* (the delimiter, e.g., , or \t),
    c. *header* (row number to use for column names)
  - *numpy:* Fast numerical computation and array operations; Python's mathematical foundation.
- **Visualization**
  - *matplotlib.pyplot:* Basic module for creating and customizing graphs and plots.
  - *seaborn:* Creates better-looking, informative statistical visualizations quickly.
- **Statistical Analysis**
  - *scipy.stats:* For advanced statistical tests and probability functions.
  - *chi2_contingency, pearsonr, etc:* Specific tests used to confirm relationships found during EDA. pearsonr checks for a straight-line relationship between two numerical variables (correlation); chi2_contingency checks if two categories (like Gender and Churn) are related; and mannwhitneyu/kruskal are used for specialized tests when your data isn't perfectly distributed.
    Parameters: These functions typically require two data arrays/variables (e.g., x and y for correlation, or a contingency table for Chi-Square) to perform the test.
- **Outlier Treatment & Preprocessing**
  - *Winsorizer:* For handling outliers (extreme data points) by setting a maximum and minimum limit for a feature extreme values, so that the outliers are pulled back to a more reasonable maximum or minimum value without being completely thrown away.
    Parameters:
    a. *capping_method* (e.g., 'gaussian', 'iqr', 'quantile')
    b. *tail* (e.g., 'both', 'left', or 'right').

c. The main parameter is the *percentile* or *multiplier* (e.g., fold for IQR or Z-score) used to define the cap limit.

- *StandardScaler:* To standardize numerical data (mean 0, variance 1) so features are balanced and helps models perform better, especially those sensitive to feature scale.

- **Modeling & Evaluation**
  - *train_test_split:* Splits data into Training, Validation, and Testing sets. This ensures you have separate, clean sets of data for teaching the model, tuning its settings, and giving it a final, honest performance grade.

    Parameters:

    a. *test_size (e.g., 0.2 or 20%): Defines the proportion of data to allocate to the test set.*
    b. *random_state (e.g., 42):* Ensures the split is the same every time you run the code (reproducibility).
    c. *stratify (y):* Ensures the split maintains the same proportion of the target variable (y) in all three sets, which is crucial for imbalanced datasets like churn analysis.

  - *GridSearchCV:* Automates the process of finding the absolute best settings (called hyperparameters) for machine learning model. It works by exhaustively testing every combination of parameters you provide and telling you which combination results in the highest performance score.

    Parameters:

    a. *estimator:* The model you want to tune (e.g., RandomForestClassifier()).
    b. *param_grid*: A dictionary listing all the parameter names and the list of values you want to test for each parameter.
    c. *cv (Cross-Validation):* Defines how many folds (subsets) to split the training data into for robust testing (e.g., cv=5).

  - *sklearn.metrics:* To grade model's performance. It gives an essential scores like accuracy (how often the model is right), precision, recall (how well it finds actual churners), the F1-score, and the confusion matrix, which together give a complete picture of how successful the predictions are.
  - *LogisticRegression: Simple, linear classification model (baseline).*

    Parameters:

a. *penalty (e.g., 'l1', 'l2', 'elasticnet'):* Specifies the regularization type to prevent overfitting.
b. *C:* The inverse of regularization strength (smaller C means stronger regularization).
c. *solver:* The algorithm used for optimization (e.g., 'liblinear', 'lbfgs').

- *RandomForestClassifier:* Powerful ensemble model for stable classification. It works by combining the predictions of hundreds of individual decision trees (ensemble method) to get a final, robust result that is usually much better than any single tree.
  Parameters:
  a. *n_estimators:* The number of trees to build in the forest (e.g., 100, 200). More trees generally improve accuracy but increase computation time.
  b. *max_depth:* The maximum depth allowed for each individual decision tree. This controls overfitting; smaller numbers prevent deep trees from memorizing the data.
  c. *criterion (e.g., 'gini', 'entropy'):* The function used to measure the quality of a split.
- *XGBClassifier*: The most accurate boosting model, often used for top results. It uses a sophisticated technique called Gradient Boosting to iteratively fix the errors of previous trees, leading to a highly optimized and very accurate final prediction.
  Parameters:
  - *n_estimators:* Similar to Random Forest, this is the number of boosting rounds/trees.
  - *learning_rate (e.g., 0.01 to 0.3):* Controls how much the model learns in each boosting step. A smaller value usually requires more n_estimators but leads to better generalization.
  - *max_depth:* Controls the complexity of each individual tree.

- Utility
  - *joblib:* For efficiently saving and loading trained ML models.
  - *tabulate:* Prints data in a neat table format to the console.
  - *warnings.filterwarnings('ignore'):* Hides warning messages to keep the output clean.

4. Code Breakdown

- Importing Libraries

```
# === Data Manipulation ===
import pandas as pd
import numpy as np

# === Visualization ===
import matplotlib.pyplot as plt
import seaborn as sns

# === Statistical Analysis ===
import scipy.stats as stats
from scipy.stats import pearsonr, mannwhitneyu, kruskal,
chi2_contingency

# === Outlier Treatment ===
from feature_engine.outliers import Winsorizer

# === Preprocessing ===
from sklearn.preprocessing import (
    StandardScaler,
)

# === Model Selection & Evaluation ===
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.metrics import (
    accuracy_score,
    precision_score,
    recall_score,
    f1_score,
    classification_report,
    confusion_matrix
)

# === Machine Learning Models ===
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from xgboost import XGBClassifier

# === Utility ===
from tabulate import tabulate
import os

import joblib
```

- Ignore Warning

```
import warnings
warnings.filterwarnings('ignore')
```

Functions to control how warnings are handled, in this case The argument 'ignore' sets the action for all subsequent warnings to be ignored, meaning they will not be displayed in the console or log.

- Color Palette Configuration

```
colors = ["#2c3e50", "#34495e", "#7f8c8d", "#e74c3c", "#c0392b"]
ordered_colors = [colors[0], colors[3], colors[2], colors[1],
colors[4]]
sns.set_palette(colors)
```

To defines a list of five hexadecimal color codes (colors) and then creates an alternative color sequence (ordered_colors) by rearranging those colors using list indexing. The final line, sns.set_palette(colors), utilizes the Seaborn library to apply the colors list as the global default color palette for all subsequent data visualizations, ensuring a consistent and customized look across the charts.

- Sets pandas display options for better output formatting

```
# Sets pandas to display all columns without truncation
pd.set_option("display.max_columns", None)
# Sets pandas to display floating point numbers with 2 decimal
places
pd.set_option('display.float_format', lambda x: '%.2f' % x)
# Sets pandas display width to 500 characters
pd.set_option('display.width', 500)
```

- Load Data

```
# read file CSV
data = pd.read_csv('Customer-Churn-Records.csv')
# show 5 rows
data.head()
```

- Data Exploration

```
# Display a random sample of 5 rows from the DataFrame
data.sample(5)
# Display a summary of the DataFrame structure (Dtypes, non-null
counts, etc.)
data.info()
# Check and count the number of missing (NaN) values per column
data.isna().sum()
# Check and count the total number of fully duplicated rows
data.duplicated().sum()
```

- Exploratory Data Analysis

This Exploratory Data Analysis (EDA) is the process of getting to know the data closely. The goal is to uncover what is hidden beneath the numbers, such as general patterns, relationships between variables, and data quality issues (like missing data or strange/outlier values). By performing EDA, we ensure the dataset is clean and we have a strong basic understanding before we start building any predictive models.

a. Univariate Analysis

- Removing Irrelevant Columns

```
irrelevant_columns = ['RowNumber', 'CustomerId', 'Surname']
data = data.drop(columns=irrelevant_columns,
errors='ignore')
```

This code is performing a data cleaning step by removing columns identified as irrelevant for machine learning or statistical analysis. Specifically, the columns 'RowNumber', 'CustomerId', and 'Surname' are dropped because 'RowNumber' is just a sequence identifier, 'CustomerId' is a unique but statistically meaningless ID, and 'Surname' (name) is assumed to have no influence on the outcome being modeled (such as customer churn). The code ensures that the DataFrame data is cleaned of these non-predictive features.

- Split numerical and categorical columns

```
numerical_cols = []   # Initialize an empty list to store
the names of true numerical (continuous) columns
categorical_cols = [] # Initialize an empty list to store
the names of categorical (object/discrete) columns

# Iterate through every column name in the 'data' DataFrame
for col in data.columns:
    # Check if the column's data type is a standard integer
(int64) or float (float64)
    if data[col].dtype in ['int64', 'float64']:
        unique_values = data[col].nunique()  # Count the
number of unique values in the numerical column

        # Check if the count of unique values is less than
or equal to 5
        if unique_values <= 5:
            # If yes (numerical but few unique values),
treat it as categorical (discrete/ordinal)
            categorical_cols.append(col)
        else:
            # If no (numerical and many unique values),
treat it as a true numerical (continuous) column
            numerical_cols.append(col)

    else:
        # If the data type is non-numerical (e.g., 'object'
or 'bool'), automatically classify as categorical
        categorical_cols.append(col)

print("Numerical Columns:", numerical_cols)    # Print the
final list of columns classified as numerical
print("Categorical Columns:", categorical_cols) # Print the
final list of columns classified as categorical
```

- Numerical Features Analysis & Analyze Distribution of Numerical Features

```
# Display summary statistics (count, mean, std, min, max,
quartiles) for all identified numerical columns.
data[numerical_cols].describe()

# Define a function to generate detailed distribution plots
(Histogram, Box Plot, Q-Q Plot) for numerical variables.
def plot_distributions(data, ordered_colors):

    # Assume the input DataFrame 'data' contains only the
    numerical columns for analysis.
    numeric_columns = data.columns

    # Extend the provided color list to ensure there are
    enough colors for every numerical column.
    # This prevents running out of colors if the number of
    columns exceeds the initial list length.
    extended_colors = (ordered_colors *
    (len(numeric_columns) // len(ordered_colors) +
    1))[:len(numeric_columns)]

    # Set the plotting style for aesthetics (using a clean
    white grid) and context (font size).
    plt.style.use('seaborn-v0_8-whitegrid')
    sns.set_context("notebook", font_scale=1.2)

    # Loop through each numerical column to create a
    dedicated set of plots.
    for i, col in enumerate(numeric_columns):

     # Create a figure with 1 row and 3 subplots for
     Histogram, Box Plot, and Q-Q Plot.
     # Gridspec is used to adjust the relative widths of
     the subplots for better visual balance.
     fig, axes = plt.subplots(1, 3, figsize=(18, 5),
     gridspec_kw={'width_ratios': [2, 1, 2]})
     # Add a bold title for the entire figure, positioned
     slightly above the plots.
     fig.suptitle(f'Univariate Analysis of {col}',
     fontsize=16, weight='bold', y=1.05)

     # --- 1. Histogram (axes[0]) ---
     # Plot the distribution histogram with a Kernel
     Density Estimate (KDE) line.
     sns.histplot(data[col], kde=True,
     color=extended_colors[i], ax=axes[0], bins=30)
     axes[0].set_title(f'{col} Histogram', fontsize=14)
     axes[0].set_xlabel(col, fontsize=12)
     axes[0].set_ylabel('Frequency', fontsize=12)

     # Calculate and draw vertical lines for Mean (dashed
     green) and Median (solid purple).
     mean_val = data[col].mean()
     median_val = data[col].median()
     axes[0].axvline(mean_val, color='green', linestyle='-
     -', label=f'Mean: {mean_val:.2f}')
     axes[0].axvline(median_val, color='purple',
     linestyle='-', label=f'Median: {median_val:.2f}')
```

```
            axes[0].legend(fontsize=10)

            # --- 2. Box Plot (axes[1]) ---
            # Plot the box plot to visualize quartiles, median,
            and outliers.
            # 'flierprops' customizes the appearance of outliers
            (red circles).
            sns.boxplot(y=data[col], ax=axes[1],
            color=extended_colors[i], flierprops=dict(marker='o',
            markerfacecolor='red', markersize=8))
            axes[1].set_title(f'{col} Box Plot', fontsize=14)
            axes[1].set_ylabel(col, fontsize=12)

            # --- 3. Q-Q Plot (axes[2]) ---
            # Plot the Quantile-Quantile plot against a normal
            distribution ('dist="norm"').
            # The 'rvalue=True' option displays the correlation
            coefficient, assessing normality.
            stats.probplot(data[col], dist="norm", plot=axes[2] ,
            rvalue=True)
            axes[2].set_title(f'{col} Q-Q Plot', fontsize=14)
            axes[2].set_xlabel('Theoretical Quantiles',
            fontsize=12)
            axes[2].set_ylabel('Sample Quantiles', fontsize=12)

            # Calculate the Skewness and Kurtosis (measures of
            distribution shape).\
            skewness = stats.skew(data[col])
            kurtosis = stats.kurtosis(data[col])

            # Create a text box within the figure to display the
            calculated statistics.
            stats_text = f'{col} Skewness: {skewness:.2f}\n{col}
            Kurtosis: {kurtosis:.2f}'
            fig.text(0.01, 0.95, stats_text, fontsize=12,
            bbox=dict(facecolor='white', alpha=0.8,
            edgecolor='black'))

            # Adjust subplot parameters for tight layout to
            prevent overlap.
            plt.tight_layout()
            # Display the generated figure.
            plt.show()

# Execute the function, passing the subset of numerical data
and the custom color palette.
plot_distributions(data[numerical_cols], ordered_colors)
```

- Categorical Features Analysis & Analyze Distribution of Categorical Features

```
# Define a function to generate pie charts and frequency
tables for all input categorical features.
def plot_categorical_features(data):
```

```
# Get the list of column names (categorical features)
from the input DataFrame.
categorical_cols = data.columns.to_list()

# Set the overall plotting style to 'seaborn-v0_8' for
aesthetics.
plt.style.use('seaborn-v0_8')
# Define a custom color palette using specific hex
codes (blue, red, green, yellow, purple).
custom_palette = ['#3498db', '#e74c3c', '#2ecc71',
'#f1c40f', '#9b59b6']
# Set the custom palette as the default for Seaborn
plots (though Matplotlib's plt.pie is used here).
sns.set_palette(custom_palette)

# Iterate through each categorical column.
for col in categorical_cols:
    # Get the unique categories (labels) and their
    counts.
    labels = data[col].value_counts().keys()
    counts = data[col].value_counts().values
    # Create formatted strings combining the count and
    its percentage for pie chart labels.
    percentages = [f'{count} ({(count / len(data) *
    100):.1f}%)' for count in counts]

    # --- 1. Generate Pie Chart ---
    plt.figure(figsize=(6, 4)) # Create a new figure
    for the pie chart.
    plt.pie(counts, labels=percentages, # Use the
    combined count/percentage string as labels.
    shadow=True,# Highlight the largest slice (index
    0) with a slight "explosion."
    explode=[0.1 if i == 0 else 0 for i in
    range(len(labels))],
    startangle=90, # Start the first slice at the top
    (90 degrees).
    textprops={'fontsize': 12, 'weight': 'bold'}) #
    Style the text labels.

    plt.title(f'Distribution of {col}',
    fontsize=14,pad=20)
    plt.axis('equal') # Ensures the pie chart is drawn
    as a circle.
    plt.tight_layout()
    plt.show() # Display the pie chart.

    # --- 2. Generate Frequency Table ---
    value_counts = data[col].value_counts() # Ge the
    raw count for each category.
    # Get the relative frequency (percentage) for each
    category, rounded to 1 decimal place.
    relative_freq =
    (data[col].value_counts(normalize=True) *
    100).round(1)

    # Create a pandas DataFrame for structured display.
```

```
            table_data = pd.DataFrame({
                'Value': value_counts.index,
                'Count': value_counts.values,
                'Relative Frequency (%)': relative_freq.values
            })

            # Print the summary table using the 'tabulate'
            library for neat formatting (psql style).
            print(tabulate(table_data, headers='keys',
            tablefmt='psql', showindex=False))
            print("\n") # Add a blank line for separation.

    # Execute the function, passing the subset of the original
    data containing only categorical columns.
    plot_categorical_features(data[categorical_cols])
```

b. Bivariate Analysis

- Analyze the relationship between CreditScore and Age

```
    # Create a new Matplotlib figure and set its dimensions to
    7x5 inches.
    plt.figure(figsize=(7, 5))

    # Generate a scatter plot with a linear regression line
    (regplot).
    sns.regplot(data=data, x='CreditScore', y='Age',
    # Customize the scatter points: 60% transparency and size
    20.
    scatter_kws={'alpha': 0.6, 's': 20},
    # Customize the regression line: set color to red and line
    width (lw) to 2.
    line_kws={'color': 'red', 'lw': 2})

    # Set the title of the plot, making it bold and clearly
    identifying the variables.
    plt.title('CreditScore vs Age', fontsize=16,
    fontweight='bold')
    # Label the X-axis.
    plt.xlabel('CreditScore', fontsize=12)
    # Label the Y-axis.
    plt.ylabel('Age', fontsize=12)
    # Add a light, dashed grid to the plot for easier data
    inspection.
    plt.grid(True, linestyle='--', alpha=0.6)
    # Remove the top and right borders (spines) for a cleaner
    aesthetic.
    sns.despine()
    # Display the generated plot.
    plt.show()

    # Calculate the Pearson correlation coefficient and the
    corresponding p-value.
    # This statistically measures the strength and direction of
    the linear relationship.
```

```
correlation, p_value = pearsonr(data['CreditScore'],
data['Age'])
# Print the results, formatted to three decimal places for
neatness.
print(f'Pearson Correlation: {correlation:.3f}, p-value:
{p_value:.3f}')
```

- Analyze the relationship between Age and Exited

```
# Create a new Matplotlib figure and set its dimensions to
6x4 inches.
plt.figure(figsize=(6, 4))

# Generate a boxplot comparing the 'Age' distribution (Y-
axis) based on 'Exited' status (X-axis).
# This visually shows if the median age and spread differ
between customers who stayed (0) and those who churned (1).
sns.boxplot(data=data, x='Exited', y='Age')

# Set the title of the plot.
plt.title('Age vs Exited')
# Label the X-axis (0: Retained, 1: Churned).
plt.xlabel('Exited')
# Label the Y-axis.
plt.ylabel('Age')
# Display the plot.
plt.show()

# Separate the 'Age' data into two distinct groups based on
the 'Exited' status.
exited_0 = data[data['Exited'] == 0]['Age'] # Ages of
customers who did NOT exit/churn (Exited = 0).
exited_1 = data[data['Exited'] == 1]['Age'] # Ages of
customers who DID exit/churn (Exited = 1).

# Perform the Mann-Whitney U test (a non-parametric test).
# This test determines if there is a statistically
significant difference in the distribution of ages
# between the two independent groups (Exited=0 and
Exited=1).
stat, p_value = mannwhitneyu(exited_0, exited_1)
# Print the test statistic and the p-value, formatted to
three decimal places.
print(f'Mann-Whitney U Statistic: {stat:.3f}, p-value:
{p_value:.3f}')
```

- Analyze the relationship between Balance vary and Geography

```
# Create a new Matplotlib figure and set its dimensions to
6x4 inches.
plt.figure(figsize=(6, 4))

# Generate a bar plot using Seaborn.
# X-axis: 'Geography' (the categorical groups: France,
Spain, Germany).
```

```
# Y-axis: 'Balance'.
# estimator='mean': The height of each bar represents the
average (mean) Balance for that geographical region.
sns.barplot(data=data, x='Geography', y='Balance',
estimator='mean')

# Set the title of the plot.
plt.title('Mean Balance by Geography')
# Label the X-axis.
plt.xlabel('Geography')
# Label the Y-axis.
plt.ylabel('Mean Balance')
# Display the plot.
plt.show()

# Prepare data for the statistical test by separating the
'Balance' data into distinct lists
# for each unique 'Geography' group (e.g., one list for
France Balances, one for Spain, etc.).
groups = [data[data['Geography'] == g]['Balance'] for g in
data['Geography'].unique()]
# Perform the Kruskal-Wallis H test (a non-parametric
test).
# This tests the null hypothesis that the median Balance is
the same across all three geographical groups.
# The asterisk (*) unpacks the list of 'groups' so the
function can accept them as separate arguments.
stat, p_value = kruskal(*groups)
# Print the test statistic and the p-value, formatted to
three decimal places.
print(f'Kruskal-Wallis Statistic: {stat:.3f}, p-value:
{p_value:.3f}')
```

- Analyze the relationship between EstimatedSalary relation and Exited

```
# Create a new Matplotlib figure and set its dimensions to
6x4 inches.
plt.figure(figsize=(6, 4))

# Generate a boxplot using Seaborn.
# X-axis: 'Exited' status (0: Stayed, 1: Left).
# Y-axis: 'EstimatedSalary'.
# This visually compares the salary distribution between
customers who stayed and those who left.
sns.boxplot(data=data, x='Exited', y='EstimatedSalary')

# Set the title of the plot.
plt.title('Estimated Salary vs Exited')
# Label the X-axis, explicitly defining the binary values
(0 and 1).
plt.xlabel('Exited (0: Stayed, 1: Left)')
# Label the Y-axis.
plt.ylabel('Estimated Salary')
# Customize the X-axis tick labels to be more readable
(replacing 0 and 1 with 'Stayed' and 'Left').
plt.xticks([0, 1], ['Stayed', 'Left'])
```

```
# Display the plot.
plt.show()

# Separate the 'EstimatedSalary' data into two distinct
groups based on the 'Exited' status.
exited_0 = data[data['Exited'] == 0]['EstimatedSalary'] #
Salaries of customers who did NOT exit/churn (Exited = 0).
exited_1 = data[data['Exited'] == 1]['EstimatedSalary'] #
Salaries of customers who DID exit/churn (Exited = 1).

# Perform the Mann-Whitney U test (a non-parametric
statistical test).
# This test determines if there is a statistically
significant difference in the distribution of salaries
# between the two independent groups (Stayed and Left).
stat, p_value = mannwhitneyu(exited_0, exited_1)
# Print the test statistic and the p-value, formatted to
three decimal places.
print(f'Mann-Whitney U Statistic: {stat:.3f}, p-value:
{p_value:.3f}')
```

- Analyze the relationship between Gender and Exited

```
# Create a new Matplotlib figure and set its dimensions to
6x4 inches.
plt.figure(figsize=(6, 4))

# Generate a contingency table (cross-tabulation) of Gender
and Exited status.
# The 'plot(kind='bar', stacked=True)' method then
immediately generates a stacked bar chart from this table.
# Each bar represents a gender, and the segments show the
count of 'Exited' status (0/1) within that gender.
pd.crosstab(data['Gender'],
data['Exited']).plot(kind='bar', stacked=True)

# Set the title of the plot.
plt.title('Gender vs Exited')
# Label the X-axis (Gender).
plt.xlabel('Gender')
# Label the Y-axis (Count of customers).
plt.ylabel('Count')
# Add a legend to explain the 'Exited' status segments (0
and 1).
plt.legend(title='Exited')
# Display the plot.
plt.show()

# --- Statistical Test ---

# Generate the raw contingency table needed for the Chi-
Square test.
contingency_table = pd.crosstab(data['Gender'],
data['Exited'])
# Perform the Chi-Square test of independence.
```

```
# This test determines if there is a statistically
significant association between Gender and Exited status.
# It returns the test statistic (chi2), the p-value,
degrees of freedom (dof), and expected frequencies.
chi2, p_value, dof, expected =
chi2_contingency(contingency_table)
# Print the results, formatted to three decimal places.
print(f'Chi-Square Statistic: {chi2:.3f}, p-value:
{p_value:.3f}')
```

- Analyze the relationship between Exited relation and IsActiveMember

```
# Create a new Matplotlib figure and set its dimensions to
6x4 inches.
plt.figure(figsize=(6, 4))

# Generate a grouped count plot using Seaborn.
# X-axis: 'IsActiveMember' (0 or 1).
# hue='Exited': Bars are split (grouped) based on the
customer's churn status (Exited=0 or Exited=1).
# palette='viridis': Uses the 'viridis' color scheme for
the bars.
sns.countplot(data=data, x='IsActiveMember', hue='Exited',
palette='viridis')

# Set the title of the plot.
plt.title('Relationship between Exited and IsActiveMember')
# Label the X-axis, explicitly defining the binary values
(0 and 1).
plt.xlabel('IsActiveMember (0: No, 1: Yes)')
# Label the Y-axis (Count of customers).
plt.ylabel('Count')
# Customize the X-axis tick labels to be more readable
(replacing 0 and 1 with 'Inactive' and 'Active').
plt.xticks([0, 1], ['Inactive', 'Active'])
# Add a legend, customizing its title and labels for
clarity (Stayed and Left).
plt.legend(title='Exited', labels=['Stayed', 'Left'])
# Display the plot.
plt.show()

# --- Statistical Test ---

# Generate the raw contingency table (cross-tabulation)
needed for the Chi-Square test.
contingency_table = pd.crosstab(data['IsActiveMember'],
data['Exited'])
# Perform the Chi-Square test of independence.
# This test determines if there is a statistically
significant association between active status and churn
status.
# It returns the test statistic (chi2), the p-value,
degrees of freedom (dof), and expected frequencies.
chi2, p_value, dof, expected =
chi2_contingency(contingency_table)
# Print the results, formatted to three decimal places.
```

```
print(f'Chi-Square Statistic: {chi2:.3f}, p-value:
{p_value:.3f}')
```

- Analyze the relationship between Tenure and IsActiveMember

```
# Create a new Matplotlib figure and set its dimensions to
6x4 inches.
plt.figure(figsize=(6, 4))

# Generate a boxplot using Seaborn.
# X-axis: 'IsActiveMember' status (0 or 1).
# Y-axis: 'Tenure'.
# This visually compares the tenure distribution between
active and inactive customers.
sns.boxplot(data=data, x='IsActiveMember', y='Tenure')

# Set the title of the plot.
plt.title('Tenure vs IsActiveMember')
# Label the X-axis.
plt.xlabel('IsActiveMember')
# Label the Y-axis.
plt.ylabel('Tenure')
# Display the plot.
plt.show()

# Separate the 'Tenure' data into two distinct groups based
on the 'IsActiveMember' status.
active_0 = data[data['IsActiveMember'] == 0]['Tenure'] #
Tenures of customers who are NOT active (IsActiveMember =
0).
active_1 = data[data['IsActiveMember'] == 1]['Tenure'] #
Tenures of customers who ARE active (IsActiveMember = 1).

# Perform the Mann-Whitney U test (a non-parametric
statistical test).
# This test determines if there is a statistically
significant difference in the distribution of tenure
# between the two independent groups (Inactive and Active
members).
stat, p_value = mannwhitneyu(active_0, active_1)
# Print the test statistic and the p-value, formatted to
three decimal places.
print(f'Mann-Whitney U Statistic: {stat:.3f}, p-value:
{p_value:.3f}')
```

- Analyze the relationship between Satisfaction Score and Exited

```
# Create a new Matplotlib figure and set its dimensions to
6x4 inches.
plt.figure(figsize=(6, 4))

# Generate a boxplot using Seaborn.
# X-axis: 'IsActiveMember' status (0 or 1).
# Y-axis: 'Tenure'.
```

```
# This visually compares the tenure distribution between
active and inactive customers.
sns.boxplot(data=data, x='IsActiveMember', y='Tenure')

# Set the title of the plot.
plt.title('Tenure vs IsActiveMember')
# Label the X-axis.
plt.xlabel('IsActiveMember')
# Label the Y-axis.
plt.ylabel('Tenure')
# Display the plot.
plt.show()

# Separate the 'Tenure' data into two distinct groups based
on the 'IsActiveMember' status.
active_0 = data[data['IsActiveMember'] == 0]['Tenure'] #
Tenures of customers who are NOT active (IsActiveMember =
0).
active_1 = data[data['IsActiveMember'] == 1]['Tenure'] #
Tenures of customers who ARE active (IsActiveMember = 1).

# Perform the Mann-Whitney U test (a non-parametric
statistical test).
# This test determines if there is a statistically
significant difference in the distribution of tenure
# between the two independent groups (Inactive and Active
members).
stat, p_value = mannwhitneyu(active_0, active_1)
# Print the test statistic and the p-value, formatted to
three decimal places.
print(f'Mann-Whitney U Statistic: {stat:.3f}, p-value:
{p_value:.3f}')
```

c. Multivariate Analysis

- Analyze interactions between Balance, Geography, and Gender on Exited?

```
# Create a contingency table (cross-tabulation) of three
variables: Geography, Gender, and Exited status.
# The 'index' creates rows grouped by both Geography and
Gender (e.g., France-Male, France-Female, etc.).
# The 'columns' are the Exited statuses (0 and 1).
contingency_table = pd.crosstab(index=[data['Geography'],
data['Gender']], columns=data['Exited'])

# Perform the Chi-Square test of independence on the 2D
contingency table.
# This test determines if there is a statistically
significant association between the combined
# Geography-Gender factor and the Exited status.
chi2, p_value, dof, expected =
chi2_contingency(contingency_table)
# Print the results, formatted to three decimal places.
print(f'Chi-Square Statistic: {chi2:.3f}, p-value:
{p_value:.3f}')
```

```
# Generate a pair plot (matrix of scatter plots and
histograms) for a subset of columns.
sns.pairplot(data[['CreditScore', 'Tenure', 'Satisfaction
Score', 'Exited']],
# Use the 'Exited' status (0 or 1) to color-code the points
in the scatter plots and histograms.
hue='Exited',
# Apply the 'coolwarm' color palette for visualization.
palette='coolwarm')
# Display the generated pair plot matrix.
plt.show()
```

- Analyze interactions between CreditScore, Tenure, and Satisfaction Score on Exited?

```
# Generate a pair plot (matrix of scatter plots and
histograms) for a subset of columns.
sns.pairplot(data[['CreditScore', 'Tenure', 'Satisfaction
Score', 'Exited']],
# Use the 'Exited' status (0 or 1) to color-code the points
in the scatter plots and histograms.
# This is crucial for visually distinguishing customers who
stayed from those who churned.
hue='Exited',
# Apply the 'coolwarm' color palette for visualization.
palette='coolwarm')
# Display the generated pair plot matrix.
plt.show()
```

- Data Preprocessing and Cleaning

```
# Create the feature matrix (X) by dropping the target variable
('Exited') from the DataFrame.
# X will contain all the input variables (predictors) used to train
the model.
X = data.drop(columns=['Exited'])

# Create the target vector (y) containing only the variable the
model is meant to predict.
# 'Exited' is the outcome (e.g., churn status) the model will learn
to classify.
y = data['Exited']

# Define the list of numerical columns where outlier treatment
(Winsorization) will be applied.
# These columns are typically those identified during EDA as having
extreme values (outliers).
numeric_cols = ['CreditScore', 'Age', 'Tenure', 'Balance',
'EstimatedSalary', 'Point Earned']

# Instantiate the Winsorizer object for outlier capping.
# capping_method='iqr': Uses the Interquartile Range (IQR) method
to determine outlier boundaries.
# tail='both': Applies the capping to both the lower and upper
tails (outliers at both ends of the distribution).
```

```
# fold=1.5: Sets the multiplier for the IQR. Outliers are typically
defined as values outside the range
# [Q1 - 1.5*IQR, Q3 + 1.5*IQR], which is the standard boundary for
boxplots.
# variables=numeric_cols: Specifies which columns should be
processed.
winsor = Winsorizer(capping_method='iqr', tail='both', fold=1.5,
variables=numeric_cols)

# Fit the Winsorizer to the training data (X) to calculate the
actual capping limits (thresholds).
# Then, transform the data by replacing all values outside the
calculated boundaries
# with the boundary values themselves. The result is stored in the
new DataFrame, X_winsorized.
X_winsorized = winsor.fit_transform(X)
```

a. Apply Transformations

```
# Apply the log(1+x) transformation to the 'Age' column.
# Purpose: To reduce the positive (right) skewness (skew of
1.01) in the 'Age' distribution,
# making it closer to a normal (Gaussian) distribution, which
benefits many models.
X_winsorized['Age'] = np.log1p(X_winsorized['Age'])

# Apply the log(1+x) transformation to the 'Balance' column.
# Purpose: To stabilize the variance and handle the zero-
inflation (many zero values) present in 'Balance'.
# This transformation is preferred over simple log(x) because it
correctly handles zero values
# (log(0) is undefined, but log1p(0) is 0).
X_winsorized['Balance'] = np.log1p(X_winsorized['Balance'])
```

b. Encode Categorical Variables

```
# One-hot encode the specified nominal categorical columns
('Geography', 'Gender', 'Card Type').
# pd.get_dummies creates new binary columns (0 or 1) for each
unique value within these features.
# drop_first=True is crucial for avoiding multicollinearity: it
drops the first category's column
# (e.g., 'France'), as its information is redundant and can be
inferred when all other category columns are 0.
X_encoded = pd.get_dummies(X_winsorized, columns=['Geography',
'Gender', 'Card Type'], drop_first=True)

# Ordinal Encode the 'Satisfaction Score' column.
# astype('category') converts the column type for efficient
encoding.
# .cat.codes converts the categories into numerical codes
(integers 0, 1, 2, ...).
# This is appropriate for 'Satisfaction Score' because the
categories have an inherent order (ordinal data).
X_encoded['Satisfaction Score'] = X_encoded['Satisfaction
Score'].astype('category').cat.codes
```

- Train-test split

```
# --- [Split 1] Initial Split: Training/Validation Pool (80%) and
Final Test Set (20%) ---

# Use train_test_split to separate 20% of the data for the final,
unseen Test Set.
# X_train_val (80%): Data used for training and tuning.
# X_test (20%): Data reserved for the final, unbiased model
performance evaluation.
# random_state=42 ensures the split is reproducible.
X_train_val, X_test, y_train_val, y_test =
train_test_split(X_encoded, y, test_size=0.2, random_state=42)

# --- [Split 2] Secondary Split: Validation Set (10% of total) from
Training Pool (80% of total) ---

# This step splits the remaining 80% (X_train_val) into the final
Training set and Validation set.
# test_size=0.125 is used because 0.125 * 80% = 10% of the original
total data.
# X_train_temp (70% of total): The final data used for training the
model.
# X_val (10% of total): Used for hyperparameter tuning and early
stopping.
X_train_temp, X_val, y_train_temp, y_val = train_test_split(
    X_train_val, y_train_val, test_size=0.125, random_state=42
)

# NOTE: X_train_temp is the final 70% Training set used for model
fitting.
# Resulting Data Proportions: Training (70%), Validation (10%),
Test (20%)

# Print the shape (rows, columns) of the final 70% Training set.
# This data will be used to fit the model parameters.
print(f"X_train_temp shape: {X_train_temp.shape}")

# Print the shape (rows, columns) of the 10% Validation set.
# This data will be used for tuning hyperparameters and development
testing.
print(f"X_val shape: {X_val.shape}")

# Print the shape (rows, columns) of the 20% reserved Test set.
# This data is saved for the final, unbiased evaluation of the
model.
print(f"X_test shape: {X_test.shape}")

# --- 1. Scaling Numerical Features ---

# Initialize the StandardScaler object. This scaler transforms data
to have a mean of 0 and a standard deviation of 1.
scaler = StandardScaler()
```

```
# FIT and TRANSFORM the Training set's numerical columns.
# 'fit' calculates the mean and standard deviation from
X_train_temp, and 'transform' applies the scaling.
X_train_scaled = scaler.fit_transform(X_train_temp[numeric_cols])

# TRANSFORM the Validation set's numerical columns using the
statistics (mean/std) learned from the Training set.
# Crucially, we use 'transform' only, preventing data leakage from
the validation set into the training process.
X_val_scaled = scaler.transform(X_val[numeric_cols])

# TRANSFORM the Test set's numerical columns using the statistics
learned from the Training set.
# This also prevents data leakage, ensuring an unbiased evaluation.
X_test_scaled = scaler.transform(X_test[numeric_cols])

# --- 2. Recreating DataFrames (for combining) ---

# Convert the scaled NumPy arrays back into pandas DataFrames,
preserving column names and the original index.
X_train_scaled_df = pd.DataFrame(X_train_scaled,
columns=numeric_cols, index=X_train_temp.index)
X_val_scaled_df = pd.DataFrame(X_val_scaled, columns=numeric_cols,
index=X_val.index)
X_test_scaled_df = pd.DataFrame(X_test_scaled,
columns=numeric_cols, index=X_test.index)

# --- 3. Final Feature Concatenation ---

# Concatenate (combine) the scaled numerical columns with the
unscaled categorical columns.
# axis=1 ensures columns are combined side-by-side.
# The categorical columns are obtained by dropping the numerical
ones from the original sets.
X_train_final = pd.concat([X_train_scaled_df,
X_train_temp.drop(columns=numeric_cols)], axis=1)
X_val_final = pd.concat([X_val_scaled_df,
X_val.drop(columns=numeric_cols)], axis=1)
X_test_final = pd.concat([X_test_scaled_df,
X_test.drop(columns=numeric_cols)], axis=1)

# --- 4. Verification ---

# Print the final shapes to confirm successful processing and
dimension consistency.
print(f'Train shape: {X_train_final.shape}, Val shape:
{X_val_final.shape}, Test shape: {X_test_final.shape}')
```

- Model Implementation
  a. Define Models within a Function (for clean, reusable code).

```
# Initialize a global dictionary to store the performance
metrics of different models.
model_metrics = {}
```

```python
# Define a function to streamline the training, prediction, and
evaluation process.
def train_and_evaluate(model, X_train, y_train, X_val, y_val,
model_name):
    # --- 1. Training and Prediction ---

     # Train the machine learning model using the training data
     (X_train, y_train).
    model.fit(X_train, y_train)
     # Use the trained model to make predictions on the
     validation feature set (X_val).
    y_val_pred = model.predict(X_val)

    # --- 2. Metric Calculation ---

    # Calculate standard classification evaluation metrics by
    comparing true labels (y_val)
    # with the model's predictions (y_val_pred).
    accuracy = accuracy_score(y_val, y_val_pred)
    precision = precision_score(y_val, y_val_pred)
    recall = recall_score(y_val, y_val_pred)
    f1 = f1_score(y_val, y_val_pred)

    # --- 3. Storing Metrics ---

    # Store the calculated metrics in the global 'model_metrics'
    dictionary
    # using the provided 'model_name' as the key.
    model_metrics[model_name] = {
        'Accuracy': accuracy,
        'Precision': precision,
        'Recall': recall,
        'F1-Score': f1
    }

    # --- 4. Displaying Results ---

    # Generate a comprehensive classification report (including
    support, macro avg, weighted avg).
     report = classification_report(y_val, y_val_pred,
     output_dict=True)
    # Convert the dictionary output into a pandas DataFrame for
    neat formatting.
    report_df = pd.DataFrame(report).transpose()

     # Print the report using the 'tabulate' library for a
     professional, table-like output.
     print(f"\n{model_name} Classification Report on Validation
     Set:")
     print(tabulate(report_df, headers='keys', tablefmt='psql',
     floatfmt='.3f'))

    # Return the trained model object and the F1-Score (often
    the main metric used for comparison).
    return model, f1
```

b. Hyperparameter Tuning menggunakan GridSearchCV

- Logistic Regression

```
# Initialize the Logistic Regression model. random_state=42
ensures reproducibility.
log_reg = LogisticRegression(random_state=42)

# Define the grid of hyperparameters to search over.
# 'C': Inverse of regularization strength (smaller C = stronger
regularization).
# 'max_iter': Maximum number of iterations for the solver to
converge.
param_grid_log = {'C': [0.01, 0.1, 1, 10], 'max_iter': [100,
200]}

# Initialize GridSearchCV to systematically search for the best
combination of parameters.
# log_reg: The model being tuned.
# param_grid_log: The parameters to test.
# cv=5: Uses 5-fold cross-validation on the training data.
# scoring='f1': The metric used to judge the best model during
the search (F1-score is chosen for balance).
# n_jobs=-1: Uses all available CPU cores for faster
computation.
grid_log = GridSearchCV(log_reg, param_grid_log, cv=5,
scoring='f1', n_jobs=-1)

# Fit the Grid Search object to the primary training data
(X_train_temp).
# This executes the cross-validation across all parameter
combinations defined in param_grid_log.
grid_log.fit(X_train_temp, y_train_temp)

# Retrieve the model instance that yielded the highest F1-score
during the Grid Search.
best_log = grid_log.best_estimator_

# Evaluate the best model on the Validation set (X_val_final,
y_val) using the custom function.
# The model is trained on the full training set (X_train_final)
before final evaluation.
best_log_f1 = train_and_evaluate(best_log, X_train_final,
y_train_temp, X_val_final, y_val, 'Logistic Regression')

# Print the specific hyperparameters that resulted in the
highest F1-score during the search.
print(f"Best Logistic Regression Params:
{grid_log.best_params_}")
```

- Random Forest Classifier

```
# Initialize the Random Forest model. random_state=42 ensures
reproducibility.
rf = RandomForestClassifier(random_state=42)
```

```
# Define the grid of hyperparameters to search over.
# 'n_estimators': Number of decision trees in the forest (50
or 100).
# 'max_depth': Maximum depth of the individual trees (None =
unlimited, 10, or 20).
# 'min_samples_split': Minimum number of samples required to
split an internal node (2 or 5).
param_grid_rf = {'n_estimators': [50, 100], 'max_depth':
[None, 10, 20], 'min_samples_split': [2, 5]}

# Initialize GridSearchCV to systematically search for the
best combination of parameters.
# rf: The model being tuned.
# param_grid_rf: The parameters to test.
# cv=5: Uses 5-fold cross-validation on the training data.
# scoring='f1': The metric used to judge the best model during
the search (F1-score is chosen for balance).
# n_jobs=-1: Uses all available CPU cores for faster
computation.
grid_rf = GridSearchCV(rf, param_grid_rf, cv=5, scoring='f1',
n_jobs=-1)

# Fit the Grid Search object to the primary training data
(X_train_final).
# This executes the cross-validation across all parameter
combinations defined in param_grid_rf.
grid_rf.fit(X_train_final, y_train_temp)

# Retrieve the model instance that yielded the highest F1-
score during the Grid Search.
best_rf = grid_rf.best_estimator_

# Evaluate the best model on the Validation set (X_val_final,
y_val) using the custom function.
# The model is trained on the full training set
(X_train_final) before final evaluation.
best_rf_f1 = train_and_evaluate(best_rf, X_train_final,
y_train_temp, X_val_final, y_val, 'Random Forest')

# Print the specific hyperparameters that resulted in the
highest F1-score during the search.
print(f"Best Random Forest Params: {grid_rf.best_params_}")
```

- XGBoost

```
# Initialize the XGBoost Classifier.
# random_state=42 ensures reproducibility.
# eval_metric='logloss' specifies the metric to monitor during
boosting (common for classification).
xgb = XGBClassifier(random_state=42, eval_metric='logloss')

# Define the grid of hyperparameters to search over.
# 'learning_rate': Controls the step size shrinkage to prevent
overfitting.
```

```
# 'max_depth': Maximum depth of each tree (complexity
control).
# 'n_estimators': Number of boosting stages (number of trees).
param_grid_xgb = {'learning_rate': [0.01, 0.1], 'max_depth':
[3, 5], 'n_estimators': [50, 100]}

# Initialize GridSearchCV to systematically search for the
best combination of parameters.
# xgb: The model being tuned.
# param_grid_xgb: The parameters to test.
# cv=5: Uses 5-fold cross-validation on the training data.
# scoring='f1': The metric used to judge the best model during
the search (F1-score is chosen for balance).
# n_jobs=-1: Uses all available CPU cores for faster
computation.
grid_xgb = GridSearchCV(xgb, param_grid_xgb, cv=5,
scoring='f1', n_jobs=-1)

# Fit the Grid Search object to the primary training data
(X_train_final).
# This executes the cross-validation across all parameter
combinations defined.
grid_xgb.fit(X_train_final, y_train_temp)

# Retrieve the model instance that yielded the highest F1-
score during the Grid Search.
best_xgb = grid_xgb.best_estimator_

# Evaluate the best model on the Validation set (X_val_final,
y_val) using the custom function.
# The model is trained on the full training set
(X_train_final) before final evaluation.
best_xgb_f1 = train_and_evaluate(best_xgb, X_train_final,
y_train_temp, X_val_final, y_val, 'XGBoost')

# Print the specific hyperparameters that resulted in the
highest F1-score during the search.
print(f"Best XGBoost Params: {grid_xgb.best_params_}")
```

- Final Model

```
# Create a pandas DataFrame from the global dictionary
'model_metrics'.
# .T (transpose) is used to switch the rows (model names) and
columns (metrics) for a better display format.
metrics_df = pd.DataFrame(model_metrics).T
# Print the comprehensive table of all calculated metrics
(Accuracy, Precision, Recall, F1) for all models.
print("\nValidation Metrics Table:")
# Use the 'tabulate' library to print the DataFrame in a neat,
psql-style table format.
print(tabulate(metrics_df, headers='keys', tablefmt='psql',
floatfmt='.3f'))

# --- Visualization ---
```

```
# Create a Matplotlib figure for the visualization.
plt.figure(figsize=(8, 5))
# Generate a horizontal bar plot comparing the 'F1-Score' of
each model.
sns.barplot(x='F1-Score', y=metrics_df.index, data=metrics_df,
palette='coolwarm')
# Set the title, x-axis label, and y-axis label.
plt.title('F1-Scores of Models on Validation Set')
plt.xlabel('F1-Score')
plt.ylabel('Model')
# Display the plot.
plt.show()

# --- Model Selection ---

# Find the name of the model that achieved the highest 'F1-
Score' in the 'model_metrics' dictionary.
# The 'max' function uses a lambda key to iterate through the
dictionary and compare F1-Scores.
best_model_name = max(model_metrics, key=lambda k:
model_metrics[k]['F1-Score'])

# Retrieve the actual best model object (e.g., the tuned
XGBoost model) using its name as the key.
# It assumes 'best_log', 'best_rf', and 'best_xgb' variables
hold the tuned model objects.
best_model = {'Logistic Regression': best_log, 'Random
Forest': best_rf, 'XGBoost': best_xgb}[best_model_name]

# Print the name and the F1-Score of the chosen best model,
which will be used for final testing.
print(f"\nBest Model: {best_model_name} with F1-Score:
{model_metrics[best_model_name]['F1-Score']:.5f}")
```

- Implement and Test Final Model (using the best parameters found).

```
# --- 1. Prediction ---

# Use the selected 'best_model' (e.g., the best tuned XGBoost) to
predict the target variable
# on the completely unseen final test set features (X_test_final).
y_test_pred = best_model.predict(X_test_final)

# --- 2. Metric Calculation ---

# Calculate the final set of performance metrics by comparing the
true test labels (y_test)
# against the model's predictions (y_test_pred).
test_accuracy = accuracy_score(y_test, y_test_pred)
test_precision = precision_score(y_test, y_test_pred)
test_recall = recall_score(y_test, y_test_pred)
test_f1 = f1_score(y_test, y_test_pred)

# --- 3. Displaying Results ---
```

```
# Create a pandas DataFrame to neatly organize the final calculated
metrics.
test_metrics = pd.DataFrame({
    'Metric': ['Accuracy', 'Precision', 'Recall', 'F1-Score'],
    'Value': [test_accuracy, test_precision, test_recall, test_f1]
})
# Print the final performance table using 'tabulate' for
professional, formatted output.
print("\nTest Metrics Table for Best Model:")
print(tabulate(test_metrics, headers='keys', tablefmt='psql',
floatfmt='.5f'))

# --- 1. Classification Report ---

# Generate a comprehensive classification report (including
precision, recall, f1-score, and support
# for both class 0 and class 1) for the final test predictions.
test_report = classification_report(y_test, y_test_pred,
output_dict=True)
# Convert the report (which is a dictionary) into a pandas
DataFrame and transpose it
# for a cleaner table layout (metrics as columns, classes as rows).
test_report_df = pd.DataFrame(test_report).transpose()
# Print the formatted classification report table for the best
model on the test set.
print(f"\n{best_model_name} Classification Report on Test Set:")
print(tabulate(test_report_df, headers='keys', tablefmt='psql',
floatfmt='.3f'))

# --- 2. Confusion Matrix Plot ---

# Calculate the Confusion Matrix (CM) by comparing the true test
labels (y_test)
# against the model's predictions (y_test_pred).
cm = confusion_matrix(y_test, y_test_pred)
# Create a new Matplotlib figure for the plot.
plt.figure(figsize=(6, 4))
# Generate a heatmap visualization of the Confusion Matrix using
Seaborn.
# annot=True: Displays the number inside each cell.
# fmt='d': Formats the numbers as integers (decimal).
# cmap='Blues': Uses a blue color scheme.
# cbar=False: Hides the color bar.
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', cbar=False)
# Set the title, x-axis label ('Predicted'), and y-axis label
('Actual').
plt.title(f'Confusion Matrix for {best_model_name} on Test Set')
plt.xlabel('Predicted')
plt.ylabel('Actual')
# Display the plot.
plt.show()
```

- Model saving (pkl)

```
import joblib
```

```
# Import the 'joblib' library, which is highly efficient for saving
and loading large Python objects,
# especially NumPy arrays and scikit-learn model objects.

# Save the trained 'best_model' object to a file named 'model.pkl'.
# 'joblib.dump()' serializes the model object, converting it into a
byte stream.
# The '.pkl' (pickle) extension is the conventional format used for
saving serialized Python objects.
joblib.dump(best_model, 'model.pkl')
```

```
import joblib
# Import the 'joblib' library (assumed to be imported earlier).

# Save the trained 'scaler' object to a file named 'scaler.pkl'.
# The 'scaler' object contains the means and standard deviations
calculated from the training data
# during the scaling process (scaler.fit_transform(X_train)).
joblib.dump(scaler, 'scaler.pkl')
# Saving the scaler is crucial because it ensures that any new,
incoming data (like customer data
# for prediction) is transformed using the EXACT same rules and
statistics used during training.

# Create a Python list containing all column names from the final
training features DataFrame.
# This list represents the exact order and names of the features
used by the model.
training_columns_list = X_train_final.columns.tolist()

# Print the resulting list of column names. This is typically done
to verify
# that all necessary features (numerical and encoded categorical)
are present.
print(training_columns_list)
```

- Load model + generate predictions

```
import pandas as pd
import numpy as np
import joblib
import warnings
import sys

# Suppress minor warnings for cleaner output
warnings.filterwarnings('ignore')

# Sets pandas to display all columns without truncation
pd.set_option("display.max_columns", None)
# Sets pandas to display floating point numbers with 2 decimal
places
pd.set_option('display.float_format', lambda x: '%.2f' % x)
# Sets pandas display width to 500 characters
pd.set_option('display.width', 500)
```

```
# --- 1. Define Assets and New Data (Simulation) ---

# CRITICAL: The list of columns MUST MATCH the exact column names
and order used during model training (Schema Alignment)
FINAL_COLUMNS = [
    'CreditScore', 'Age', 'Tenure', 'Balance', 'EstimatedSalary',
'Point Earned',
    'NumOfProducts', 'HasCrCard', 'IsActiveMember', 'Complain',
'Satisfaction Score',
    'Geography_Germany', 'Geography_Spain', 'Gender_Male', 'Card
Type_GOLD',
    'Card Type_PLATINUM', 'Card Type_SILVER']

numeric_cols = ['CreditScore', 'Age', 'Tenure', 'Balance',
'EstimatedSalary', 'Point Earned']
categorical_cols_ohe = ['Geography', 'Gender', 'Card Type']
ordinal_col_name = 'Satisfaction Score'

# Raw data for a new customer (simulated input)
new_customer_raw = pd.DataFrame({
    'CustomerID': [15634602], # Customer ID (key)
    'CreditScore': [619], 'Age': [42], 'Tenure': [2], 'Balance':
[0],
    'EstimatedSalary': [101348.88], 'Point Earned': [464],
'Geography': ['France'],
    'Gender': ['Female'], 'NumOfProducts': [1], 'HasCrCard': [1],
    'IsActiveMember': [1], 'Complain': [1], 'Satisfaction Score':
[2],
    'Card Type': ['DIAMOND']})
# new_customer_raw

# Take only CustomerID
# Store the Customer ID separately as it is not used in the model
features
customer_id_df = new_customer_raw[['CustomerID']].copy()
# customer_id_df

# --- 2. Load Deployment Assets (Model and Scaler) ---

try:
    # Load the trained model object (e.g., XGBoost, Logistic
     Regression)
    best_model = joblib.load('model.pkl')
     # Load the fitted scaler object (StandardScaler or
    MinMaxScaler)
    scaler = joblib.load('scaler.pkl')
    training_columns = FINAL_COLUMNS
    print("Assets (Model, Scaler, Columns) loaded successfully.")
except FileNotFoundError:
    print("Error: Ensure model.pkl and scaler.pkl files are in the
    same directory.")
    sys.exit(1)

# --- 3. Preprocessing New Data (Preventing Schema Drift) ---

# Duplicate raw data and drop CustomerID (as it's not a feature)
new_data = new_customer_raw.drop(columns=['CustomerID']).copy()
```

```
# A. Log1p Transformation (Must be identical to the training
process)
new_data['Age'] = np.log1p(new_data['Age'])
new_data['Balance'] = np.log1p(new_data['Balance'])

# B. Ordinal Encoding (Converts ordinal scores to numerical
categories: 0, 1, 2, ...)
new_data[ordinal_col_name] =
new_data[ordinal_col_name].astype('category').cat.codes
# new_data

# C. Scaling Numerical Features (Apply the fitted scaler to the new
data)
new_data_scaled = scaler.transform(new_data[numeric_cols])
# Convert the scaled array back into a Pandas DataFrame
new_data_scaled_df = pd.DataFrame(new_data_scaled,
columns=numeric_cols, index=new_data.index)
# new_data_scaled_df

# D. Categorical Encoding and Concatenation
data_non_numeric = new_data.drop(columns=numeric_cols)
# Perform One-Hot Encoding (OHE) on nominal features
new_data_ohe = pd.get_dummies(data_non_numeric,
columns=categorical_cols_ohe, drop_first=True)
# Combine scaled numerical features with OHE categorical features
new_data_processed = pd.concat([new_data_scaled_df, new_data_ohe],
axis=1)
# new_data_processed

# E. Finalization: Reindex (CRITICAL STEP for Schema Alignment)
# Aligns columns to the exact list and order specified in
training_columns.
# fill_value=0 adds any missing OHE columns (if the new data
doesn't contain a specific category, like 'Geography_Germany') and
sets their value to 0.
X_predict_final =
new_data_processed.reindex(columns=training_columns, fill_value=0)

print("New customer data processed and aligned to model schema.")
print(f"Final Data Shape: {X_predict_final.shape}")

# ------------------------------------------------------------
# --- 4. Prediction Using Loaded Model ---
# ------------------------------------------------------------
try:
    # Predict the class (0 or 1)
    prediction = best_model.predict(X_predict_final)[0]

    # Predict the full probability array (e.g., [[P_Class_0,
    P_Class_1]])
    all_probas = best_model.predict_proba(X_predict_final)

    # Extract probability of Churn (Class 1) for summary
    proba = all_probas[0][1]

    # --- 5. Final Results (Summary) ---
```

```
        status = "CHURN" if prediction == 1 else "NOT CHURN (Stay)"

        print("\n--- MODEL PREDICTION RESULTS (SUMMARY) ---")
        print(f"Model ({type(best_model).__name__}) predicted:")
        print(f"Customer Status: {status}")
        print(f"Churn Probability (Class 1): {proba:.4f}")

        # ------------------------------------------------------------
        # --- 6. Create Prediction DataFrame (FINAL OUTPUT) ---
        # ------------------------------------------------------------

        # 1. Extract the probability column for Class 1 (index 1) and
    convert to a DataFrame
        predict_proba_xgb = pd.DataFrame(
            all_probas[:, 1],
            columns=['PREDICT_PROBA_RESULT']
        )

        # 2. Concatenate the Customer ID with the Probability Result
        df_final_proba =
    pd.concat([customer_id_df.reset_index(drop=True),
    predict_proba_xgb], axis=1)

    except ValueError as e:
        print(f"\nPREDICTION FAILED.")
        print(f"Error: {e}")

    # --- FINAL OUTPUT ---
    # This block prints the final DataFrame containing the CustomerID
    and the prediction probability.
    try:
        print("\n--- PREDICTION OUTPUT DATAFRAME ---")
        print(df_final_proba)
    except NameError:
        print("\nDataFrame df_final_proba could not be displayed
        because the prediction failed.")
```

1. Troubleshooting

Main Sources of Slowdowns	Detailed Explanation	Best Actions & Solutions
1. Combination Explosion	GridSearchCV tests every combination of defined hyperparameters (exhaustive search).	Use **RandomizedSearchCV** or **Bayesian Optimization** (e.g., Optuna). These test a smart/random sample of the best parameters instead of the entire grid.
2. Cross-Validation (CV) Load	Each parameter combination is repeated **K times** (folds) for validation. Example: 100 combinations × 5 folds = 500 models trained.	Reduce the number of CV folds (e.g., from 10 to 3 or 5), especially during initial tuning.

3. Low CPU Utilization	By default, scikit-learn may use only one CPU core for training, even if your machine has multiple cores.	Always set **n_jobs=-1** in GridSearchCV. This forces all CPU cores to be used in parallel, giving the largest speed boost.
4. Model Complexity	Models like Random Forest (with high n_estimators) or XGBoost take a long time to fit, which multiplies when using Grid Search.	For initial tuning, use a **subset of the training data**. After finding the best parameters, train the final model on the **full dataset**.

2. Hyperlinks & Additional Notes

- Python docs for warnings — Warning control library : https://docs.python.org/3/library/warnings.html
- Pandas Formatting option : https://pandas.pydata.org/docs/reference/api/pandas.set_option.html
- Winsorizer : https://feature-engine.trainindata.com/en/1.8.x/user_guide/outliers/Winsorizer.html
- np.log : https://medium.com/@noorfatimaafzalbutt/understanding-np-log-and-np-log1p-in-numpy-99cefa89cd30
- gridsearchCV https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html
- Pickle ML https://medium.com/@prathik.codes/a-quick-guide-to-using-pickle-files-in-machine-learning-c13bb22a2c81